2000

# Impact of Tile-Size Selection for Skewed Tiling

Yonghong Song

Zhiyuan Li
*Purdue University*, li@cs.purdue.edu

Report Number:
00-018

# IMPACT OF TILE-SIZE SELECTION FOR SKEWED TILING

Yonghong Song
Zhiyuan Li

Department of Computer Sciences
Purdue Unversity
West Lafayette, IN 47907

CSD TR #00-018
December 2000
Revised January 2001

# Impact of Tile-Size Selection for Skewed Tiling *

Yonghong Song Zhiyuan Li
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
{songyh,li}@cs.purdue.edu

**Abstract**

Tile-size selection is known to be a complex problem. This paper develops a new selection algorithm. Unlike previous algorithms, this new algorithm considers the effect of loop skewing on cache misses. It also estimates loop overhead and incorporates them into the execution cost model, which turns out to be critical to the decision between tiling a single loop level vs. tiling two loop levels. Our preliminary experimental results show a significant impact of these previously ignored issues on the execution time of tiled loops. In our experiments, we measured the cache miss rate and the execution time of five benchmark programs on a single processor and we compared our algorithm with previous algorithms. Our algorithm achieves an average speedup of 1.27 to 1.63 over all the other algorithms.

## 1 Introduction

Memory access latency has become the key performance bottleneck on modern microprocessors. In order to reduce the average memory reference latency, it is important to exploit data locality such that most memory references can be served by the fast memory, e.g. the cache, in the memory hierarchy. *Tiling* is a well-known compiler technique to enhance data locality such that more data can be reused before they are replaced from the cache [23]. Tiling transforms a loop nest by combining strip-mining and loop interchange. *Loop skewing* and *loop reversal* are often used to enable tiling [20]. Figure 1 shows SOR relaxation as an example. Figure 1(a) shows the original loop nest in SOR, and Figure 1(b) shows the tiled SOR in which loop $J$ is skewed with respect to loop $T$, and Figure 1(c) shows the tiled SOR in which loops $J$ and $I$ are skewed with respect to loop $T$.

Much of previous work on tiling applies to perfectly-nested loops only [8, 20, 21, 23]. Recently, we proposed a new technique to tile a class of imperfectly-nested loops [17, 18]. Performance of a tiled loop nest can vary dramatically with different tile sizes [9]. How to select proper tile sizes is hence an important issue. In this paper, if loop skewing is applied before tiling, such a tiling is called *skewed tiling*. *Non-skewed tiling* results if loop skewing is not necessary for tiling. All previous work tacitly assumes non-skewed tiling [4, 6, 9, 12, 16, 22]. However, such an assumption may not be true, especially for loops which perform iterative relaxation computations [17, 18]. Another important factor ignored in previous work is the loop overhead in terms of the increased instruction counts due to the increased loop levels. Further, tiling a software-pipelined loop will also

*This work is sponsored in part by National Science Foundation through grants CCR-9975309 and MIP-9610379, by Indiana 21st Century Fund, by Purdue Research Foundation, and by a donation from Sun Microsystems, Inc.

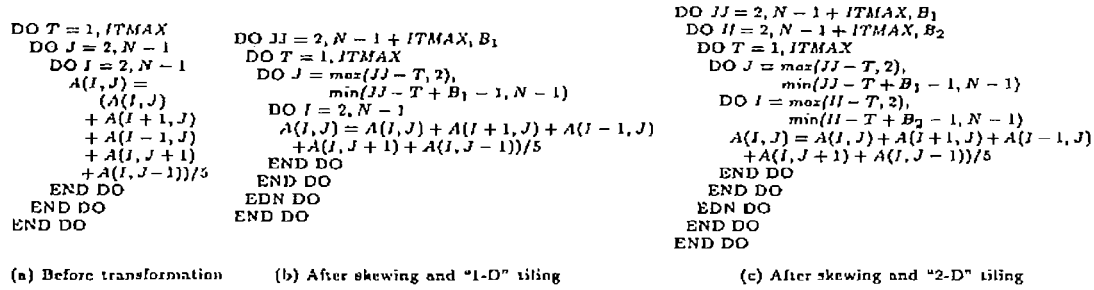



Figure 1: An example of tiling: SOR relaxation.

increase the dynamic count of load instructions. In this paper, we shall show that these previously ignored factors can have a significant effect on tile-size selection.

In our recent work [17], we present a memory cost model to estimate cache misses, assuming that only one loop level is tiled. In this paper, we present a more general scheme by considering two loop levels which may both be tiled. We present an algorithm to compute tile sizes such that during each *tile traversal*, capacity misses and self-interference misses are eliminated. Further, cross-interference misses are eliminated through array padding [15]. Given a tile size, we model the tiling cost based on both the number of cache misses and the loop overhead. To choose between tiling one loop level vs. tiling two loop levels, our algorithm computes their lowest costs and the respective tile sizes. We then choose the tiling level, and the corresponding best tile size, which yields the lowest cost. One can easily extend our discussion to higher loop levels, but such an extension does not seem useful for applications known to us.

In this paper, we consider data locality and performance enhancement on a single processor whose memory hierarchy includes cache memories at one or more levels. We have applied our tile-size selection algorithm to five numerical kernels, SOR, Jacobi, Livermore Loop No. 18 (LL18), `tomcatv` and `swim`, using a range of matrix sizes. We evaluate our algorithm on one processor of an SGI multiprocessor and on a SUN uniprocessor workstation. We compare our algorithm with TLI [3], TSS [4], LRW [9] and DAT [13]. Experiments show that our algorithm achieves a average speedup of 1.27 to 1.63 over all these previous algorithms.

In the rest of the paper, we first present a background in Section 2. We then present our memory cost model in Section 3. We model the execution time and present our tile-size selection algorithm in Section 4. We discuss related work in Section 5. In Section 6, we report experimental results and compare our algorithm with previous algorithms. Finally, we conclude in Section 7.

# 2 Background

In this section, we first define our program model and a few key parameters. We then discuss the issues of the memory hierarchy.

## 2.1 Tiling

Most of previous research on tiling addresses perfectly-nested loops only [8, 20, 21, 23]. After tiling, the loops remain perfectly-nested. In our recent work [17, 18], we perform tiling on a class of imperfectly-nested loops. Figure 2(a) shows a representative loop nest before tiling, where the $T$-loop body consists of $m$ perfectly-nested loops. The depth of each perfectly-nested inner loop is at least two. The loop bounds $L_{ij}$ and $U_{ij}$, $1 \leq i \leq m$, $j = 1, 2$, are $T$-invariant. We assume that the

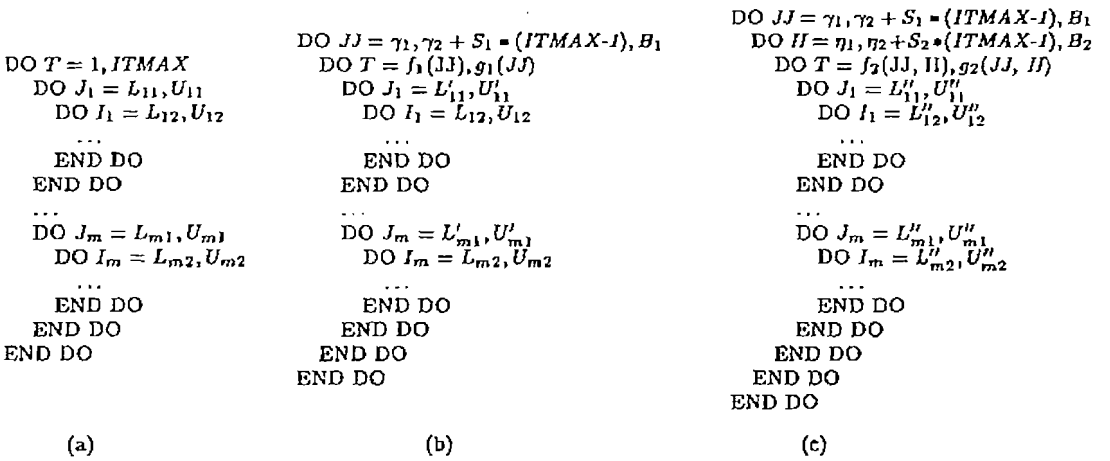



Figure 2: The program model before and after tiling

iteration space determined by $J$ and $I$ remains unchanged over different $T$-loop index values. For simplicity of presentation, we also assume that cache-line spatial locality is already fully exploited in the innermost loops except on the loop boundaries. Figure 2(b) shows the code after tiling the $J_i$ loops only (*1-D tiling*), and Figure 2(c) shows the code after tiling both $J_i$ and $I_i$ loops (*2-D tiling*). In Figures 2(b) and 2(c), the iteration subspace defined by all $J_i$ and $I_i$ loops is called a *tile*. Loop $T$ is called the *tile-sweeping* loop, and loops $JJ$ and $II$ are called the *tile-controlling* loops [20]. Each combination of $JJ$ and $II$ defines a *tile traversal*. Two tiles are said to be *consecutive* within a tile traversal if the difference of the corresponding $T$ values equals 1. In this paper, we assume the data dependences permit both 1-D and 2-D tiling. Choosing between 1-D vs. 2-D tiling will depend on the estimate of cache misses and loop overhead. As far as estimating cache misses is concerned, 1-D tiling can be viewed as a special case of 2-D tiling with the maximum tile height. However, 2-D tiling incurs higher loop overhead, which we want to take into account.

Let $\gamma_1 = min\{L_{i1} | 1 \leq i \leq m\}$, $\gamma_2 = max\{U_{i1} | 1 \leq i \leq m\}$, $\eta_1 = min\{L_{i2} | 1 \leq i \leq m\}$ and $\eta_2 = max\{U_{i2} | 1 \leq i \leq m\}$. We call $S_1$ and $S_2$ the *skewing factors* corresponding to $J_i$ and $I_i$ loops respectively. (The skewing factors are also called the *slope* in our previous work [17, 18].) If $S_1 = 0$, then loop skewing is not applied before tiling at the $J_i$ level. In this paper, we are interested only in skewed tiling at least at the $J_i$ level, thus $S_1 > 0$. $B_1$ is called the *tile width* and $B_2$ is called the *tile height*. $B_1$ and $B_2$ are called the *tile size* collectively. These parameters are used to define the bounds of the tile-controlling loops. For reference, Table 1 lists all the symbols used in this paper and their brief descriptions.

For simplicity, we assume all arrays are of two dimensions with the same column sizes. (We assume column-major storage.) Lower dimension variables can be ignored due to their lesser impact on cache misses in relaxation programs which we are interested in. Let $n_a$ be the number of two dimensional arrays for the given tiled loop nest. Within the innermost loop $I_i$, $1 \leq i \leq m$, of the untiled program in Figure 2(a), we assume array subscript patterns of $A_k(I_i + a, J_i + b)$, $1 \leq k \leq n_a$, where $a$ and $b$ are known integer constants.

## 2.2 Memory Hierarchy

The memory hierarchy includes registers, cache memories at one or more levels, the main memory and the secondary storage, as well as the TLB [7].

The TLB translates a virtual address into a physical address. The TLB has two key parameters,

Table 1: Description of symbols

| Symbol | Description | Symbol | Description |
|---|---|---|---|
| $\gamma_1$ | The minimum lower bound of all $J_i$ loops | $\gamma_2$ | The maximum upper bound of all $J_i$ loops |
| $\eta_1$ | The minimum lower bound of all $I_i$ loops | $\eta_2$ | The maximum upper bound of all $I_i$ loops |
| $S_1$ | The skewing factor for $J_i$ loops | $S_2$ | The skewing factor for $I_i$ loops |
| $B_1$ | The tile width | $B_2$ | The tile height |
| $n_a$ | The number of arrays in the given loop nest | $N$ | The array column size |
| $\gamma$ | $\gamma_2 - \gamma_1 + 1$ | $\eta$ | $\eta_2 - \eta_1 + 1$ |
| $T_c$ | The number of TLB entries | $T_b$ | The number of data elements each TLB entry can represent |
| $C_{s1}$ | The L1 cache size in the number of data elements | $C_{b1}$ | The L1 cache line size in the number of data elements |
| $C_{a1}$ | The L1 cache set associativity | $C_{s2}$ | The L2 cache size in the number of data elements |
| $C_{a2}$ | The L2 cache set associativity | $C_{b2}$ | The L2 cache line size in the number of data elements |
| $T_s$ | The TLB size in the number of data elements | $r$ | Defined in Section 3 |
| $p_1$ | The L1 cache miss penalty | $p_2$ | The L2 cache miss penalty |
| $\sigma$ | The array footprint width constrained by the TLB (see Section 4.2.3) | | |
| $n_1$ | The sum of the static number of instructions for the computation of all the $I_i$ loop bounds | | |
| $n_2$ | The sum of the static number of instructions in the $I_i$ loop bodies | | |
| $n_3$ | The sum of the static number of instructions computing the $J_i$ loop bounds | | |
| $n_4$ | The sum of the dynamic number of load instructions in the prologues and the epilogues of all software-pipelined $I_i$ loops | | |
| $n_5$ | The sum of the number of load instructions divided by the unroll factor in the software-pipelined loop bodies | | |
| $S_o$ | the iteration space defined by $\gamma_1 \le J_i \le \gamma_2$ and $\eta_1 \le I_i \le \eta_2$ in Figure 2(a) | | |
| $ITMAX$ | The maximum index value for the tile-sweeping loop | | |
| $W$ | The working-set size of the loop nest (Figure 2(a)) in the number of data elements | | |

namely the *block count* $T_c$ and the *block size* $T_b$. We call $T_s \equiv T_c T_b$ the TLB size. In this paper, $T_b$ is the size of the virtual memory represented by each TLB entry in the number of data elements. We assume a fully-associative TLB with an LRU replacement policy.

For simplicity of presentation, we consider two levels of caches in this paper, namely the L1 and L2 caches, which are common in current practice. The L1 cache has several parameters, namely the *cache size* $C_{s1}$, the *cache block size* $C_{b1}$ and the *set associativity* $C_{a1}$. $C_{s1}$ and $C_{b1}$ are measured in the number of data elements. Similarly for L2 cache, the cache size, cache block size and set associativity are $C_{s2}$, $C_{b2}$ and $C_{a2}$ respectively. The cache misses can be divided into three classes [7]: *compulsory misses*, *capacity misses* and *conflict misses*. Conflict misses can be attributed to self-interference misses of the same array and to cross-interference misses between different arrays.

# 3 A Memory Cost Model

In this section, we want to estimate the number of cache misses incurred by executing the loop nest in our program model after tiling.

Let $S_o$ represent the iteration space defined by $\gamma_1 \le J_i \le \gamma_2$ and $\eta_1 \le I_i \le \eta_2$ in Figure 2(a). (For simplicity, we also regard $S_o$ as the original iteration space defined by $J_i$ and $I_i$ loops in Figure 2(a), as if all $J_i$ loops have the same loop bounds and all $I_i$ loops have the same loop bounds.) $S_o$ is illustrated in Figure 3(a) by the rectangle enclosed by the solid lines with the height $\eta$ and the width $\gamma$. Within each tile traversal, we define the *base tile* to be a tile with $T = 1$ and an *advanced tile* to be a tile with $T > 1$. The dashed-lines in Figure 3(a) separate the base tiles of different tile traversals. The two shaded areas illustrate two different tile traversals, *tt1* and *tt2*, where each shaded rectangle with solid-line boundaries represents an advanced tile. When the tile-sweeping loop $T$ increases the index by 1, the tiles can only overlap partially.

The cache misses incurred by one tile traversal can be partitioned into those within the base tile and those within the advanced tiles. Note that only those base tiles and advanced tiles overlapping with $S_o$ will be executed, thus only they can contribute to the cache misses. In Figure 3(a), the base tile in the tile traversal *tt1* resides outside $S_o$, while the base tile in *tt2* resides within $S_o$.

We make the following two assumption in our estimation of the number of cache misses:

- **Assumption 1:** There exist no cache reuse between different tile traversals.

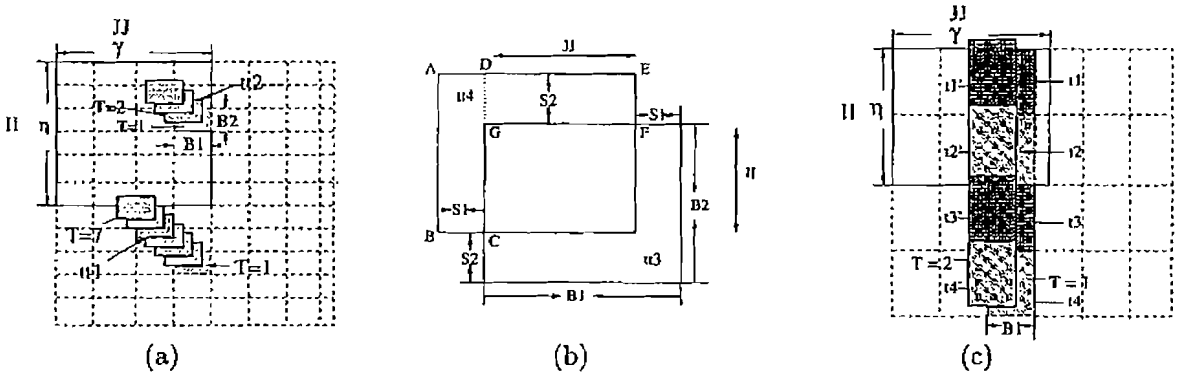



Figure 3: Illustration of tile traversal

- **Assumption 2:** $B_1 \ll \gamma$ and $B_1 \ll$ *(ITMAX-1)* $* S_1$.

Assumption 1 is reasonable if *ITMAX* is large, since it will be very likely for a tile traversal to overwrite cache lines whose old data could have been reused in the next tile traversal. Assumption 2 is reasonable because a large $B_1$ can easily cause an overflow in the TLB. As explained later in Section 4, our algorithm poses a constraint on $B_1$ such that TLB should not overflow. If the tile size ($B_1$, $B_2$) is chosen properly, there should be exactly one cache miss for each cache line accessed within a tile traversal. To be more specific, the following two properties should hold:

- **Property 1:** No capacity and self-interference misses are generated within a tile traversal.
- **Property 2:** No cross-interference misses are generated within a tile traversal.

In Section 4.2, we shall discuss how to preserve the above properties. For now, we assume they hold.

We first show how to compute the number of L1 cache misses caused by an advanced tile. Let $W$ represent the size of the data set accessed by the original loop nest in terms of the number of data elements. The average size of the data accessed by one tile is estimated to be $D = \frac{W}{\gamma\eta} * B_1 B_2$. Figure 3(b) shows two consecutive tiles, *tt3* and *tt4*, within a tile traversal, assuming that both tiles reside within $S_o$. The iteration subspace of *tt4* is produced by shifting the iteration subspace of *tt3* upwards by $S_2$ iterations and to the left by $S_1$ iterations. The L1 cache misses in *tt4* either occur in Region ABCD or in Region DEFG. The total estimated L1 cache misses equal to $(S_1 B_2 + S_2 B_1 - S_1 S_2) * \frac{W}{\gamma\eta C_{b1}}$. (This estimate may not be exact because data accessed at the lower border of Region DEFG may or may not be in the cache already.)

We then show how to accumulate the number of L1 cache misses for all the tile traversals with the same $JJ$ value. Figure 3(c) illustrates the idea. For a particular $JJ$ value, let $t_1$, $t_2$, $t_3$ and $t_4$ be the base tiles of four tile traversals, and let $t_1'$, $t_2'$, $t_3'$ and $t_4'$ be the corresponding advanced tiles when $T$ increases by 1. In this particular illustration, the number of L1 cache misses caused collectively by $t_i$ $(1 \le i \le 4)$ equals to the sum of the number of L1 cache misses caused by each individual $t_i$, that is, $\frac{WB_1}{\gamma C_{b1}}$. Note that only the tiles overlapping with $S_o$ can contribute to L1 cache misses. Similarly, the number of L1 cache misses caused by the advanced tiles $t_i'$ $(1 \le i \le 4)$ equal to the sum of the number of L1 cache misses caused by individual $t_i'$, that is, $\frac{S_1 W}{\gamma C_{b1}} + 2(B_1 - S_1) S_2 * \frac{W}{\gamma\eta C_{b1}}$. In general, the number of L1 cache misses caused by the advanced tiles with the same $JJ$ value equal to $\frac{S_1 W}{\gamma C_{b1}} + \tau(B_1 - S_1) S_2 * \frac{W}{\gamma\eta C_{b1}}$, where $\tau$ is the number of base tiles in $S_o$ for a particular $JJ$

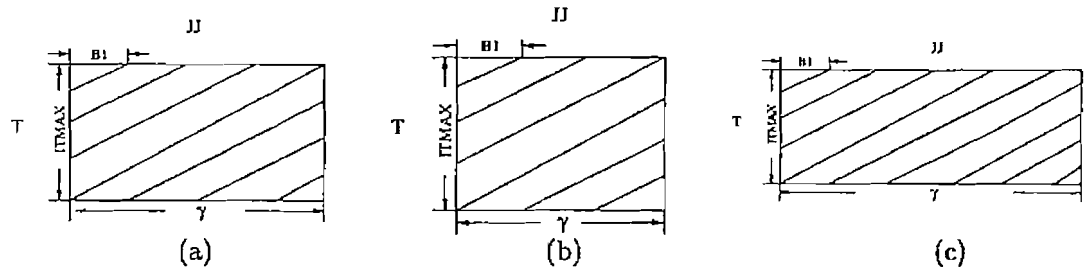



Figure 4: Calculating cache misses under different scenarios

estimated as

$$\tau = \begin{cases} \lceil \frac{\eta}{B_2} \rceil & \text{if } 1 \leq B_2 < \eta + S_2 * \text{(ITMAX-1)} \\ 0 & \text{if } B_2 = \eta + S_2 * \text{(ITMAX-1)} \end{cases}$$

The value $\eta + S_2 * \textit{(ITMAX-1)}$ is the maximum height of the iteration space after tiling. Any $B_2$ value greater than or equal to $\eta + S_2 * \textit{(ITMAX-1)}$ results in no tiling at the $I_i$ loop level.

With Assumptions 1 and 2, we can then accumulate L1 cache misses corresponding to different $JJ$ values by considering three different cases:

- **Case 1:** $\gamma = \textit{(ITMAX-1)} * S_1$.
  This case is illustrated by Figure 4(a). In this case, the tile traversals defined by $JJ \leq \gamma_1 + \gamma - \textit{NSTEP}$ will not execute to the $\textit{ITMAX}$th $T$-iteration. The tile traversal defined by $\gamma_1 + \gamma - \textit{NSTEP} < JJ \leq \gamma_1 + \gamma$ is the first to reach the $\textit{ITMAX}$th $T$-iteration. The tile traversals defined by $JJ > \gamma_1 + \gamma$ will start executing at $T > 1$. During the execution, the tile traversals defined by $JJ = \gamma_1$ will incur L1 cache misses of $\frac{WB_1}{\gamma C_{b1}}$. The tile traversals defined by $JJ = \gamma_1 + B_1$ will incur L1 cache misses of $\frac{WB_1}{\gamma C_{b1}} * 2 + \tau(B_1 - S_1)S_2 * \lceil \frac{B_1}{S_1} \rceil * \frac{W}{\gamma \eta C_{b1}}$. Hence, we have the following:

  - The L1 cache misses in all the tile traversals defined by $JJ \leq \gamma_2 - B_1$ amount to $\frac{WB_1}{\gamma C_{b1}} * (1 + 2 + \ldots + \lceil \frac{\gamma - B_1}{B_1} \rceil) + \tau(B_1 - S_1)S_2 * \frac{B_1}{S_1} * \frac{W}{\gamma \eta C_{b1}} * (1 + 2 + \ldots + \lceil \frac{\gamma - 2 * B_1}{B_1} \rceil)$.
  - The L1 cache misses in all the tile traversals defined by $\gamma_2 - B_1 < JJ \leq \gamma_2$ amount to $\frac{WB_1}{\gamma C_{b1}} * \lceil \frac{\gamma}{B_1} \rceil + \tau(B_1 - S_1)S_2 * \frac{B_1}{S_1} * \frac{W}{\gamma \eta C_{b1}} * \lceil \frac{\gamma - B_1}{B_1} \rceil$.
  - The L1 cache misses in all the tile traversals defined by $\gamma_2 < JJ$ amount to $\frac{WB_1}{\gamma C_{b1}} * (1 + 2 + \ldots + \lceil \frac{\gamma - B_1}{B_1} \rceil) + \tau(B_1 - S_1)S_2 * \frac{B_1}{S_1} * \frac{W}{\gamma \eta C_{b1}} * (1 + 2 + \ldots + \lceil \frac{\gamma - 2 * B_1}{B_1} \rceil)$.

  Adding up the three numbers of the above, the total L1 cache misses in the tiled loop nest approximate $\frac{W\gamma}{C_{b1}} * \frac{1}{B_1} + \frac{W\gamma}{C_{b1}} * \frac{S_2 \tau}{S_1 \eta}$.

- **Case 2:** $\gamma < \textit{(ITMAX-1)} * S_1$.
  This case is illustrated by Figure 4(b). Similar to the computation in Case 1, we have the following:

  - The L1 cache misses in all the tile traversals defined by $JJ \leq \gamma_2$ amount to $\frac{WB_1}{\gamma C_{b1}} * (1 + 2 + \ldots + \lceil \frac{\gamma}{B_1} \rceil) + \tau(B_1 - S_1)S_2 * \frac{B_1}{S_1} * \frac{W}{\gamma \eta C_{b1}} * (1 + 2 + \ldots + \lceil \frac{\gamma - B_1}{B_1} \rceil)$.
  - The L1 cache misses in all the tile traversals defined by $\gamma_2 < JJ \leq \textit{(ITMAX-1)} * B_1 + \gamma_1$ amount to $\frac{WB_1}{\gamma C_{b1}} * \lceil \frac{\gamma}{B_1} \rceil * \lceil \frac{\textit{(ITMAX-1)} * S_1 - \gamma}{B_1} \rceil + \tau(B_1 - S_1)S_2 * \frac{B_1}{S_1} * \frac{W}{\gamma \eta C_{b1}} * \lceil \frac{\gamma}{B_1} \rceil * \lceil \frac{\textit{(ITMAX-1)} * S_1 - \gamma}{B_1} \rceil$.

- The L1 cache misses in all the tile traversals defined by $(ITMAX\text{-}1) * B_1 + \gamma_1 < JJ$ amount to $\frac{WB_1}{\gamma C_{b1}} * (1 + 2 + \ldots + \lceil \frac{\gamma}{B_1} \rceil) + \tau(B_1 - S_1)S_2 * \frac{B_1}{S_1} * \frac{W}{\gamma\eta C_{b1}} * (1 + 2 + \ldots + \lceil \frac{\gamma - B_1}{B_1} \rceil)$.

Adding up the three numbers of the above, the total L1 cache misses in the tiled loop nest approximate $\frac{WS_1(ITMAX\text{-}1)}{C_{b1}B_1} + \frac{WS_2(ITMAX\text{-}1)\tau}{\eta C_{b1}}$.

- Case 3: $\gamma > (ITMAX\text{-}1) * S_1$.

Similar to Case 2, the total L1 cache misses in the tiled loop nest approximate $\frac{WS_1(ITMAX\text{-}1)}{C_{b1}B_1} + \frac{WS_2(ITMAX\text{-}1)\tau}{\eta C_{b1}}$.

Combining the above three cases and plugging in the estimate of $\tau$, the total number of L1 cache misses is approximately

$$\frac{WS_1(ITMAX\text{-}1)}{C_{b1}B_1} + \frac{WS_2(ITMAX\text{-}1)}{C_{b1}B_2}. \tag{1}$$

Similarly, with Properties 1 and 2 standing, the number of L2 cache misses for 2-D tiling is approximately

$$\frac{WS_1(ITMAX\text{-}1)}{C_{b2}B_1} + \frac{WS_2(ITMAX\text{-}1)}{C_{b2}B_2}. \tag{2}$$

With 1-D tiling (in Figure 2(c)), the L1 cache temporal locality is not exploited across the $T$-loop iterations. The number of L1 cache misses is approximately

$$ITMAX * \frac{W}{C_{b1}}. \tag{3}$$

The total number of cache misses for the L2 cache is approximately

$$\frac{WS_1(ITMAX\text{-}1)}{C_{b2}B_1}. \tag{4}$$

# 4 Tile-Size Selection

In this section, we first present an execution cost model for tiling with a given tile size, based on both the number of cache misses and the loop overhead. We then present our tile-size selection algorithm, followed by a running example to go through our algorithm.

## 4.1 An Execution Cost Model for Tiling

Loop tiling introduces loop overhead. To decide between 1-D tiling and 2-D tiling, the overhead of the tiled $I_i$ loops in Figure 2(c) needs to be measured. Let $n_1$ be the sum of the static number of instructions for the computation of all the $I_i$ loop bounds ($1 \leq i \leq m$). The $I_i$ loop overhead due to 2-D tiling in terms of the dynamic count of instructions, is measured approximately by

$$n_1 * \frac{ITMAX * \gamma * \eta}{B_2}. \tag{5}$$

Let $n_2$ be the sum of the static number of instructions in the $I_i$ ($1 \leq i \leq m$) loop bodies. The dynamic instruction count for the $I_i$ loop bodies is

$$n_2 * ITMAX * \gamma * \eta. \tag{6}$$

From (5) and (6), if $n_1$ and $n_2$ are approximately equal, then a small $B_2$ will introduce large loop overhead. Let $n_3$ be sum of the static number of instructions for the computation of all the $J_i$ loop bounds ($1 \leq i \leq m$). The loop overhead due to tiled $J_i$ loops can be measured by

$$n_3 * \frac{ITMAX * \gamma}{B_1}. \tag{7}$$

Enabled by scalar replacement [2], in a software-pipelined loop [1], loaded data can be reused in different iterations. The dynamic count of load instructions can hence be reduced. Let $n_4$ be the sum of the dynamic count of load instructions in the prologues and the epilogues of all the software-pipelined loops. Let $n_5$ be the sum of the number of load instructions divided by the unroll factor in the software-pipelined loop bodies. The unroll factor is one if the loop is not unrolled. The dynamic count of load instruction with 1-D tiling is approximately

$$(n_4 + n_5\gamma)\eta * ITMAX. \tag{8}$$

With 2-D tiling, the dynamic count of load instructions is approximately

$$(n_4 + n_5 B_2) * \frac{\gamma}{B_2} * \eta * ITMAX = (n_4 \frac{\gamma}{B_2} + n_5\gamma)\eta * ITMAX. \tag{9}$$

Clearly, if $n_4$ is significantly greater than $n_5$ and $B_2$ is small, then the dynamic count of load instructions with 2-D tiling can be much greater than that with 1-D tiling.

Let $p_1$ be the penalty for an L1 cache miss and $p_2$ be the penalty for an L2 cache miss. By adding the penalty due to L1 cache misses in Formula (3), the penalty due to L2 cache misses in Formula (4), the loop overhead due to tiled $J_i$ loops in Formula (7), and the dynamic count of load instructions for software-pipelined innermost loops in Formula (8), we can model the execution cost for 1-D tiling by .

$$p_1 * (ITMAX * \frac{W}{C_{b1}}) + p_2 * (\frac{WS_1(ITMAX\text{-}1)}{C_{b2}B_1}) + n_3\frac{ITMAX * \gamma}{B_1} + (n_4 + n_5\gamma)\eta * ITMAX. \tag{10}$$

In the above formula, we assume the latency of one unit of time for each instruction, including a load instruction. From (10), with 1-D tiling, we want to maximize $B_1$ (subject to Properties 1 and 2 aforementioned) such that the number of L2 cache misses is minimized. By adding the penalty due to L1 cache misses in Formula (1), the penalty due to L2 cache misses in Formula (2), the dynamic count of load instructions for software-pipelined innermost loops in Formula (9), the loop overhead due to tiled $J_i$ loops in Formula (7), and the loop overhead due to the tiled innermost loop in Formula (5), the execution cost for 2-D tiling can be modeled by

$$p_1 * (\frac{WS_1(ITMAX\text{-}1)}{C_{b1}B_1} + \frac{WS_2(ITMAX\text{-}1)}{C_{b1}B_2}) + p_2 * (\frac{WS_1(ITMAX\text{-}1)}{C_{b2}B_1} + \frac{WS_2(ITMAX\text{-}1)}{C_{b2}B_2})$$
$$+n_1 * \frac{ITMAX * \gamma * \eta}{B_2} + n_3\frac{ITMAX * \gamma}{B_1} + (n_4\frac{\gamma}{B_2} + n_5\gamma)\eta * ITMAX. \tag{11}$$

## 4.2 Tile-Size Selection Algorithm

In this section, we first discuss how to preserve Properties 1 and 2. We then present our tile-size selection algorithm.

```
Procedure EnumFPSize(C_s, C_b, N)
  for F_2 ← 1 to N do
    F_1 ← 1
    t ← (F_1 * N) mod C_s
    while ((F_2 + C_b − 1) ≤ t ≤ (C_s − F_2 − C_b + 1))
      Record (F_1, F_2)
      F_1 ← F_1 + 1
      t ← (F_1 * N) mod C_s
    end while
  end for
```

(a)

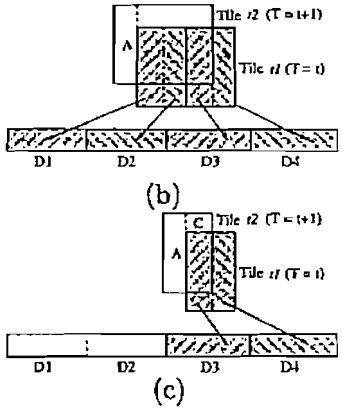



Figure 5: Procedure *EnumFPSize* and an illustration of utilizing portions of the cache by a single tile

### 4.2.1 Preserving Property 1

First, we discuss how to eliminate self-interference misses within a single tile. For any array $A_i$, let $R$ be the minimum rectangular array region which contains all the $A_i$ elements referenced within a tile $t$. We say that $A_i$'s footprint size within tile $t$ is $(F_1, F_2)$, where $F_1$ and $F_2$ are the numbers of columns and rows in $R$ respectively. We call $F_1$ ($F_2$) the *array footprint width (height)* for $A_i$ within tile $t$. Reversely, given a footprint size of $A_i$, the tile size can also be computed. Given the subscript patterns and the loop bounds, such a computation is straightforward and we omit the details. For the example of SOR (Figure 1(c)), assuming the array footprint size for $A$ to be $(\kappa_1, \kappa_2)$, the loop tile size should be $(\kappa_1 - 2, \kappa_2 - 2)$. For array $A_i$, if the footprint height $F_2$ is greater than the distance between the locations of two columns in the cache, then the columns accessed within the tile will conflict in the cache, creating self-interference misses [3]. More precisely, we have the following lemma:

**Lemma 1** Given array footprint size $(F_1, F_2)$ for any $A_i$ $(1 \leq i \leq n_a)$, a cache of size $C_s$ and cache line size $C_b$, if there exist no self-interference misses, then the distance between the starting cache locations of any two columns of $A_i$ within $F_1$ consecutive columns is either no smaller than $F_2$, or no greater than $C_s - F_2$. Conversely, there exist no self-interference misses if the distance between the starting cache locations of any two columns of $A_i$ within $F_1$ consecutive columns is either no smaller than $F_2 + C_b - 1$, or no greater than $C_s - F_2 - C_b + 1$.
**Proof** Obvious. ◇

Given a directly-mapped cache of size $C_s$ and cache line size $C_b$, and given an array column size $N$, procedure *EnumFPSize* in Figure 5(a) enumerates all the footprint sizes $(F_1, F_2)$ which incur no self-interference misses, according to Lemma 1. We say that a footprint size $(F_1, F_2)$ of $A_i$ is *maximal* if increasing either $F_1$ or $F_2$ will introduce self-interference misses for $A_i$. In general, the maximal footprint size for array $A_i$ is not unique. According to *EnumFPSize*, the maximal footprint sizes for all arrays are the same if they have the same array column sizes. Our tile-size selection scheme will enumerate all array footprint sizes which are free of self-interference misses until the sizes become maximal. The scheme estimates and compares the execution cost for different $(F_1, F_2)$ in order to get the optimal tile size.

Next, suppose the cache is not directly-mapped, and assume an LRU replacement policy.

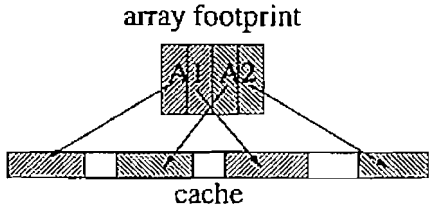



Figure 6: An illustration of padding to eliminate cross-interferences

We show that the parameter $C_s$ in procedure *EnumFPSize* should not be the whole cache size. Otherwise, self-interference misses will occur when the execution proceeds from one tile to the next. For clarity, instead of arguing formally for the general cases, we illustrate the cases of 2-way and fully-associative caches. Figure 5(b) shows two consecutive tiles *t1* and *t2*. Suppose $C_s$ equals the whole cache size in procedure *EnumFPSize* and suppose the footprint size of *t1* is maximal. Tile *t1* accesses the cache from the least-recently referenced data segment to the most-recently referenced data segment in the memory, in the order of *D1*, *D2*, *D3* and *D4* which are separated by solid lines. If the cache associativity is $C_{a1} = 2$, then *D2* and *D4* will map to the same cache sets. The data accessed in the blank rectangle A will replace segment *D2*. If the cache is fully associative, *D1* will be replaced. However, part of the old data in segment *D2* (or *D1*) could have been reused by tile *t2*. One solution to avoid the replacement of useful data is to reduce the footprint size within *t1* such that only a portion of the cache is used to compute the maximal footprint size in *EnumFPSize*. Figure 5(c) shows the case for two-way set-associative cache. In this way, the data accessed in Regions A and C will replace the cache segment *D2* and part of segment *D1*, whose old data are not reused by *t2*. The reusable data in *D3* will be kept in the cache. Using the above idea, we let $C_s = \frac{C_{a1}-1}{C_{a1}} C_{s1}$ in procedure *EnumFPSize*, for 2-way and fully-associative caches. The cases of other associativities are more complex, and they will not be discussed in this paper.

To eliminate capacity misses, the footprint size of each array $A_i$ can only be $(\lfloor \frac{F_1}{n_a} \rfloor, F_2)$, a fraction of $(F_1, F_2)$. Here, we choose to partition columns instead of rows, in order to preserve spatial locality. Assume that $(B_1^{(i)}, B_2^{(i)})$, $1 \leq i \leq n_a$, is the tile size such that the footprint size for array $A_i$ within a single tile is $(\lfloor \frac{F_1}{n_a} \rfloor, F_2)$. For 2-way and fully-associative caches, we choose the tile size for the tiled loop as $(B_1, B_2) = (min_i B_1^{(i)}, min_i B_2^{(i)})$. For directly-mapped caches, we choose $(B_1, B_2) = (min_i B_1^{(i)} - S_1, min_i B_2^{(i)} - S_2)$. One can prove that for directly-mapped, 2-way and fully-associative caches, Property 1 holds under the above treatment. For other set-associative caches, procedure *EnumFPSize* needs to be revised.

### 4.2.2 Preserving Property 2

We apply inter-array padding to eliminate cross-interference misses within a tile traversal. For simplicity of presentation, we assume that the array subscript patterns of one particular array $A_k$ cover all the array subscript patterns for all the other arrays $A_i$, $i \neq k$. The discussion in this section can be easily extended if such an assumption does not hold. Using inter-array padding, we let the starting addresses for array $A_i (1 \leq i \leq n_a)$ map to the same location in the cache as the starting address of the $(\lfloor \frac{W}{n_a} \rfloor * (i-1))$th column of array $A_1$. With such padding, cross-interference misses are eliminated within a single tile between $A_i$ and $A_j$ $(1 \leq i, j \leq n_a, i \neq j)$.

When the execution goes from one tile to the next, if the cache is directly-mapped, the newly accessed data for $A_i$ will map to cache locations previously unused in the tile traversal. If the cache is not directly-mapped, the newly accessed data for $A_i$ will map to cache locations which are

```
Input: S1, S2, Cs1, Ca1, Cb1, Cs2, Ca2, Cb2, n1, n3, n4, n5, na, N, σ (see Table 1).
Output: Tile size (B1, B2) and the transformed array declaration.
Procedure:
  if (Ca1 = 1) then
    ComputeTileSize-2D(Cs1)
    ComputeTileSize-1D(Cs2)
  else
    ComputeTileSize-2D(((Ca1 - 1)/Ca1) Cs1)
    ComputeTileSize-1D(((Ca2 - 1)/Ca2) Cs2)
  end if
  Apply inter-array padding (see Section 4.2.2).
  Return (B1, B2).

Procedure ComputeTileSize-1D(Cs)
  /* (TB1, TB2) is a temporary tile size. */
  Select the maximum tile width κ such that the footprint of one tile can fit in both the TLB and the L2 cache.
  TB1 ← κ - S1, TB2 ← η + S2 * (ITMAX-1)
  Compute the execution cost, TM, based on (10).
  if (TM < M) then B1 ← TB1, B2 ← TB2, M ← TM end if

Procedure ComputeTileSize-2D(Cs)
  /* (TB1, TB2) is a temporary tile size. */
  M ← ∞
  for F2 ← Cb1 to N do
    F1 ← 1
    t ← (F1 * N) mod Cs
    while (F1 ≤ σ or (F2 + Cb1 - 1) ≤ t ≤ (Cs - F2 - Cb1 + 1)) do
      Convert array footprint size (F1, F2) to loop tile size (TB1, TB2) (see Section 4.2.1).
      if (Ca1 = 1) then TB1 ← TB1 - S1, TB2 ← TB2 - S2 end if
      if (TB1 > 0 and TB2 > 0) then
        Compute the execution cost, TM, based on (11).
        if (TM < M) then B1 ← TB1, B2 ← TB2, M ← TM end if
      end if
      F1 ← F1 + 1
      t ← (F1 * N) mod Cs
    end while
  end for
```

Figure 7: Tile-size selection algorithm - STS

either previously unused or will not be referenced again within the current traversal. Therefore, cross-interference misses are also eliminated within a tile traversal. Figure 6 illustrates an example for $F_1 = 4$ and $n_a = 2$, where the cache is directly mapped. Here, assuming the starting address for array $A_1$ to be 0, the padded number of data items, $x$, between arrays $A_1$ and $A_2$ can be determined from

$$(size(A_1) + x) = (2 * N), \ \mathbf{mod}\ C_{s1}. \tag{12}$$

We are ready to present our tile-size selection algorithm in the next section.

### 4.2.3 Algorithm STS

Algorithm *STS* in Figure 7 selects the tile size by interleaving the operations in procedure *EnumFP-Size* with the applications of Formulas (10) and (11) which compute the execution cost. We require $B_2$ to be no smaller than the cache line size $C_{b1}$. However, we do not require $B_2$ to be a multiple of $C_{b1}$, since such a requirement does not have much benefit when execution proceeds from one tile to the next. In addition to the conditions stated in procedure *EnumFPSize*, the array footprint width $F_2$ should be no greater than $\sigma$, which is the total number of array columns representable by the TLB minus the number of newly accessed array columns when the execution proceeds from one tile to the next.

STS makes the decision between 1-D and 2-D tiling based on their execution cost. For 1-D tiling, *ComputeTileSize-1D* tries to find tile width $B_1$ such that Properties 1 and 2 are preserved on the L2 cache and that Formula (10) is minimized. For 2-D tiling, *ComputeTileSize-2D* enumerates all tile sizes which are free of self-interference misses. The tile size with the lowest execution cost is selected. Between 1-D and 2-D tiling, the scheme with the lower execution cost is chosen.

STS needs a conversion from array footprint size $(F_1, F_2)$ to loop tile size $(B_1, B_2)$, as stated in Section 4.2.1. If the resulting tile width or tile height is nonpositive, 1-D tiling is chosen.

The complexity of STS is $O(N * min(C_{s1}, \sigma)) = O(N\sigma)$. (In practice, $\sigma$ is much smaller than the L1 cache size $C_{s1}$.)

### 4.3 A Running Example

We now take SOR (Figure 1) as an example to show how STS works, assuming the following parameters: $N = 1000$, $ITMAX = 1050$, $C_{s1} = 4096$, $C_{b1} = 4$, $C_{a1} = 2$, $C_{s2} = 128 * 1024$, $C_{b2} = 16$, $C_{a2} = 2$, $T_b = 4096$ and $T_c = 48$, $n_1 = 15$, $n_3 = 15$, $n_4 = 20$, $n_5 = 3$, $p_1 = 6$, and $p_2 = 30$. Based on the array subscripts and the loop bounds, we have $S_1 = S_2 = 1$, $\gamma = \eta = 999$, $W = N * N = 1000000$ and $\sigma = 195$.

In the following, we show the steps of STS.

- Since $C_a = 2$, *ComputeTileSize-2D*$(\frac{C_{s1}}{2})$ is called, and we have $B_1 = 38$, $B_2 = 43$. The execution cost for 2-D tiling is $M = 4171464893$ units based on Formula (11).
- *ComputeTileSize-1D*$(\frac{C_{s2}}{2})$ computes $TB_1 = 63$, $TB_2 = 2048$. The execution cost for 1-D tiling is $TM = 4764840588$ units based on Formula (10). In this case, STS favors 2-D tiling over 1-D tiling with the tile size $(38, 43)$.
- No inter-array padding is applied since $n_a = 1$.

## 5 Related Work

### 5.1 Competing Tile-Size Selection Schemes

Chame and Moon present a tile size selection algorithm, called TLI, to simultaneously eliminate self-interference misses and minimize the summation of capacity misses and cross-interference misses [3]. Coleman and McKinley provide a tile size selection algorithm, TSS, based on the cache organization and the data layout [4]. TSS utilizes a gcd algorithm to exploit maximum cache utilization while eliminating all self-interference misses. Rivera and Tseng present a variation of TSS algorithm [16]. Lam *et al.* provide a tile size selection scheme, LRW, which tries to select a square tile size to eliminate the capacity and self-interference misses for a dominant array [9]. Panda *et al* present DAT, which always chooses square tile sizes and tries to minimize the interferences by padding [13]. Unlike the work in this paper, these tile-size selection algorithms do not consider the effect of loop skewing, nor do they take loop overhead into account.

### 5.2 Other Related Work

Ghosh *et al.* estimate cache misses, given a tile size, for a perfect loop nest [6]. They also informally discuss a tile-size selection scheme using matrix multiplication as the example. No formal algorithm is presented, however. They do not discuss the estimation of cache misses for imperfectly-nested loops. Therefore, we are not able to compare with their method in our experiments.

Table 2: Machine parameters

| Processors | $C_{s1}$ | $C_{b1}$ | $C_{a1}$ | $C_{s2}$ | $C_{b2}$ | $C_{a2}$ | $T_c$ | $T_b$ | $p_1$ | $p_2$ |
|---|---|---|---|---|---|---|---|---|---|---|
| Ultra II | 2K | 2 | 1 | 256K | 8 | 1 | 64 | 1K | 6 | 45 |
| R10K | 4K | 4 | 2 | 512K | 16 | 2 | 64 | 4K | 9 | 68 |

Ferrante *et al.* present an algorithm to estimate the number of distinct cache lines over a perfect loop nest [5]. Temam *et al.* derive an analytical method to estimate the number of self-interference misses [19]. Mckinley *et al.* present a simple cost model to estimate the number of cache misses [11]. These methods do not consider the effect of loop skewing.

Rivera and Tseng present several padding algorithms to eliminate cache conflict misses [15, 16]. Manjikian and Abdelrahman use *cache partitioning* to scatter arrays evenly in the cache, such that cross-interference misses are minimized [10]. We use a different padding scheme which seems more suitable for our algorithm.

# 6 Experimental Evaluation

We apply our tile-size selection algorithm STS to three numerical kernels, SOR, Jacobi and Livermore Loop No. 18 (LL18), and two SPEC benchmarks, `tomcatv` and `swim`. We use reference inputs for `tomcatv` and `swim`. For SOR, Jacobi and LL18, we declare $N \times N$ double precision arrays, with randomly chosen $N$ based on a random number generator [14] with the following formula

$$z_{n+1} = (16807 z_n) \bmod 2147483647. \tag{13}$$

Assuming that the array sizes under consideration range from $r_0$ to $r_1$, we select 200 array sizes, $a_n$, such that

$$a_n = r_0 + (z_n \bmod (r_1 - r_0)), 1 \leq n \leq 200. \tag{14}$$

We use $z_1 = 9$ in all our experiments. Note that it would be too time-consuming to exhaustly test all array sizes within the range in our experiments.

We run the test programs on a SUN Ultra II uniprocessor workstation and on one MIPS R10K processor of an SGI Origin 2000 multiprocessor, with the tile sizes selected by five different algorithms, namely, STS, TLI [3], TSS [4], LRW [9] and DAT [13]. In order to handle several equally-important arrays, we make an obviously necessary modification on the original TSS and LRW algorithms such that the value of the initial tile size will meet the working set constraint. We also modify the TLI algorithm such that only the cache size divided by the number of equally-important arrays is used to compute the tile sizes which are free of self-interference misses. If any algorithm decides to choose the whole array column as the tile height, then we let $B_2 = \eta + S_2 * (\text{ITMAX-1})$ and tile the $J_i$ loops only (Figure 2(b)).

Table 2 lists the machine parameters for the Ultra II and the R10K, assuming the size of an array element of 8 bytes. To accommodate the competition between instructions and data in the L2 cache both on the Ultra II and on the R10K, we only tries to utilize 95% of the total L2 cache capacity. We use the machine counters on the R10K and the Ultra II to measure the cache miss rate. Currently, we obtain the values of $n_1$, $n_3$, $n_4$ and $n_5$ by examining the assembly code of the original program. A backend compiler can easily obtain such numbers.

On the R10K, the untiled codes are compiled using the native compiler with the "-O3" optimization switch set. On the R10K, we found that compiling the tiled code with the "-O2" switch can sometimes run faster than that with the "-O3" switch, regardless of the tile-size selection

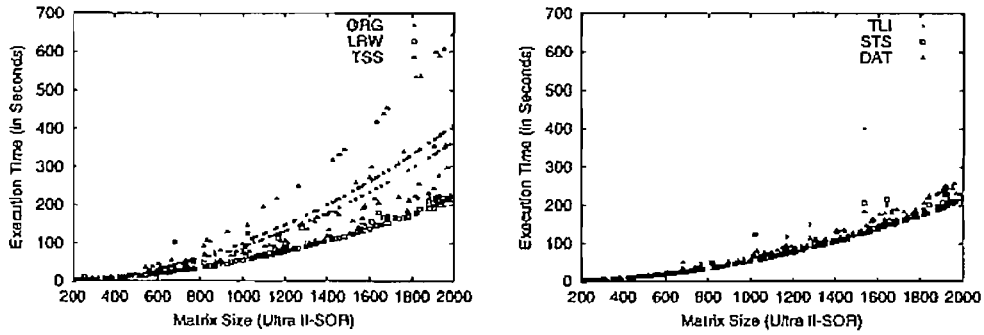



Figure 8: Execution time of SOR for various schemes on the Ultra II

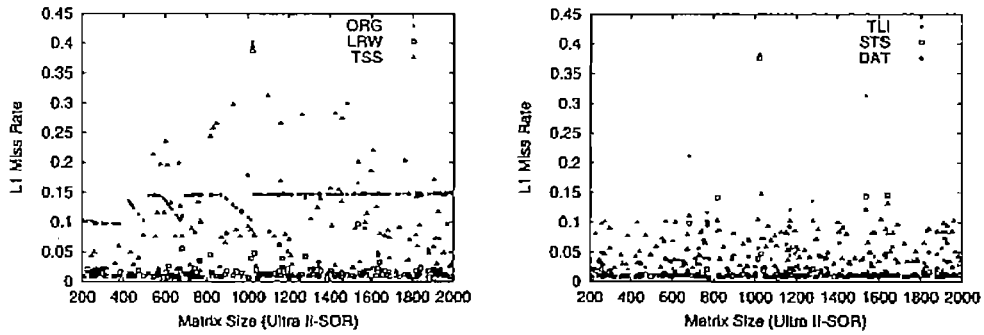



Figure 9: L1 cache miss rate of SOR for various schemes on the Ultra II

schemes. Therefore, we compile the tiled code with "-O2" or "-O3" depending on which produces shorter execution time. For all the tile-size selection schemes, we switch off loop tiling for the native compiler on the R10K when we compile the tiled source programs (with for both 1-D and 2-D tiling). We switch off prefetching on the R10K when we compile 2-D tiled source codes since prefetching may increase cross-interference misses for smaller tile height $B_2$. We also switch off common block reorganization since the tile size selection algorithms already take care of memory layout. On the Ultra II, both the untiled and the tiled codes are compiled using the native compiler with the "-fast -xchip=ultra2 -xarch=v8plusa -fsimple=2" optimization switch, which is recommended by the vendor.

`The SOR kernel`

We fix $ITMAX$ to 1050 and randomly choose 200 array sizes ranging from 200 to 2000, i.e., $(r_0, r_1) = (200, 2000)$ in Equation (14). The skewing factors are $S_1 = S_2 = 1$. We have $n_1 = n_3 = 11$, $n_4 = 9$ and $n_5 = 3$ on the R10K and $n_1 = n_3 = 22$, $n_4 = 34$, $n_5 = 4$ on the Ultra II. Table 3 summarizes the average speedup by STS over other schemes, average L1 and L2 cache miss rates for SOR on both the Ultra II and the R10K. The execution time is averaged by geometric mean, and the cache miss rates are averaged by arithmetic mean of cache miss rates for individual array size. Specifically, Figures 8 and 11 show the execution time for various schemes on the Ultra II and on the R10K respectively. Figures 9 and 10 show the L1 cache and L2 cache miss rates respectively on the Ultra II. Figures 12 and 13 show the L1 cache and L2 cache miss rates respectively on the R10K.

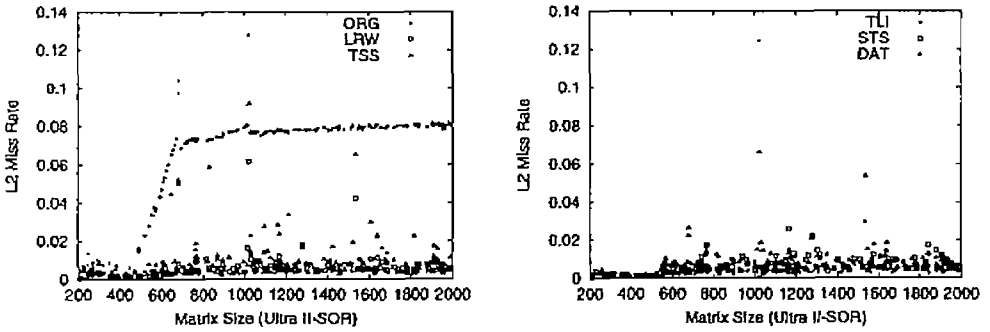



Figure 10: L2 cache miss rate of SOR for various schemes on the Ultra II

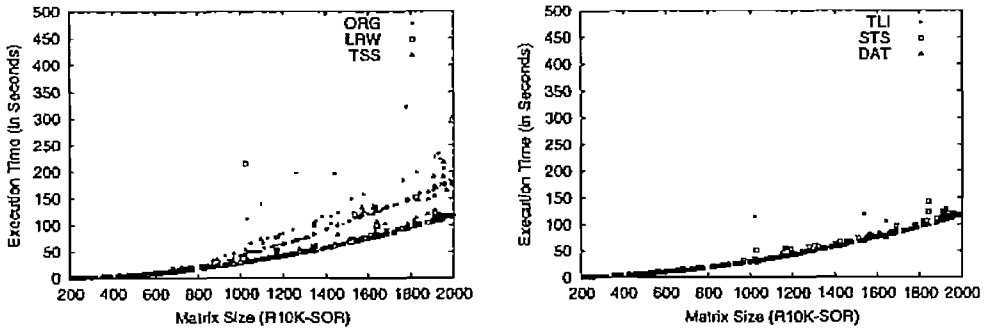



Figure 11: Execution time of SOR for various schemes on the R10K

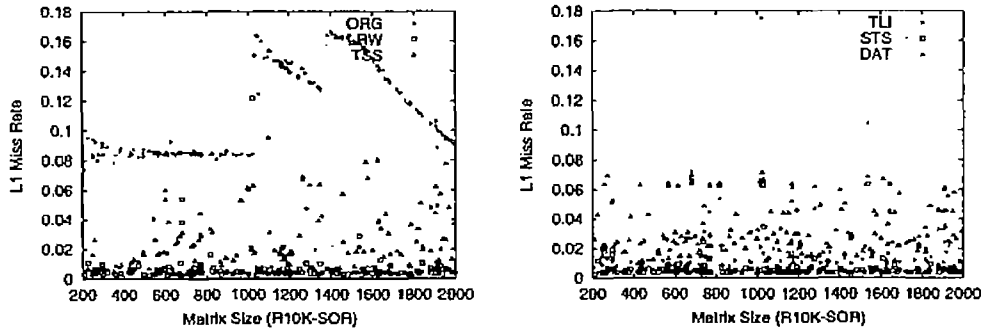



Figure 12: L1 cache miss rate of SOR for various schemes on the R10K

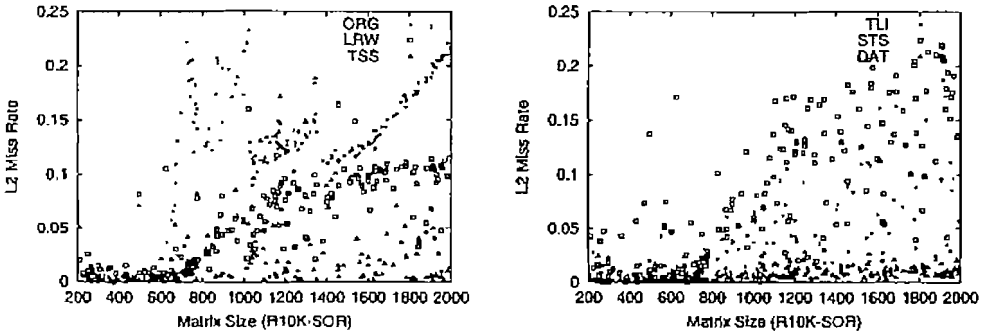



Figure 13: L2 cache miss rate of SOR for various schemes on the R10K

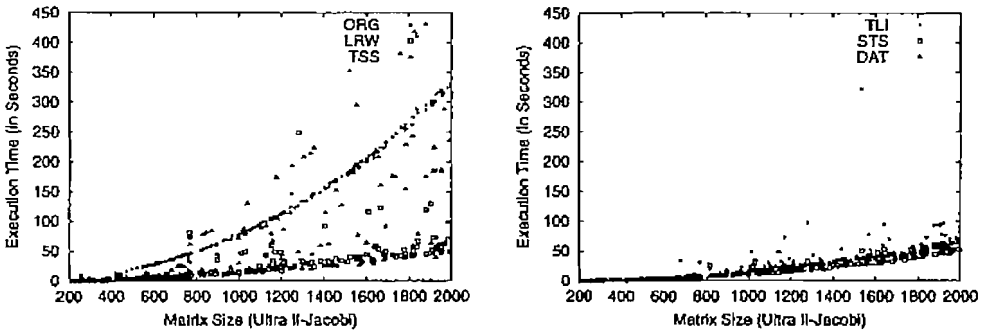



Figure 14: Execution time of Jacobi for various schemes on the Ultra II

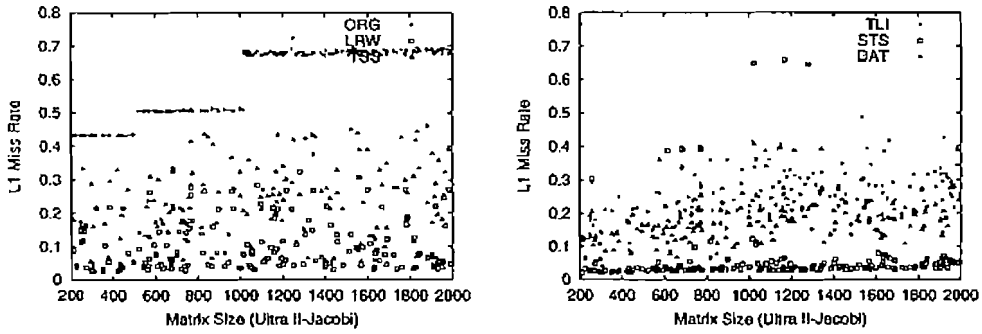



Figure 15: L1 cache miss rate of Jacobi for various schemes on the Ultra II

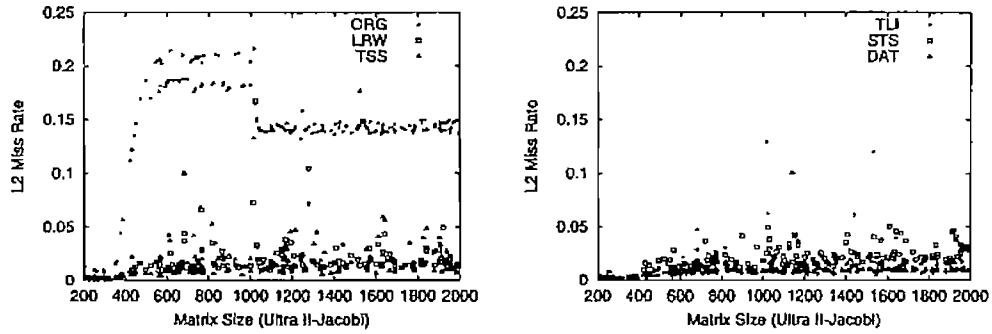



Figure 16: L2 cache miss rate of Jacobi for various schemes on the Ultra II

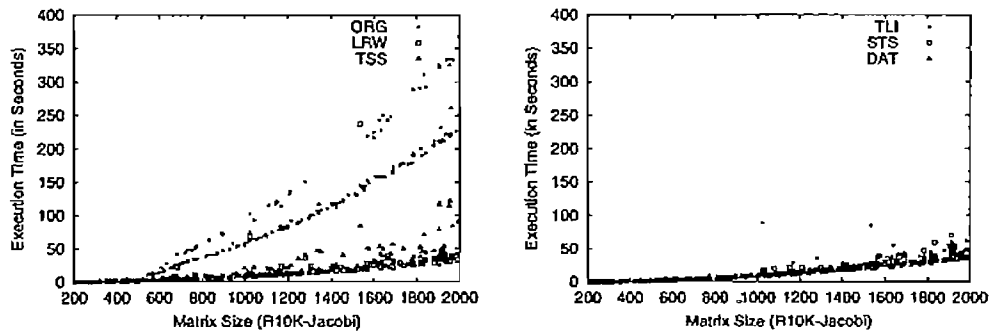



Figure 17: Execution time of Jacobi for various schemes on the R10K

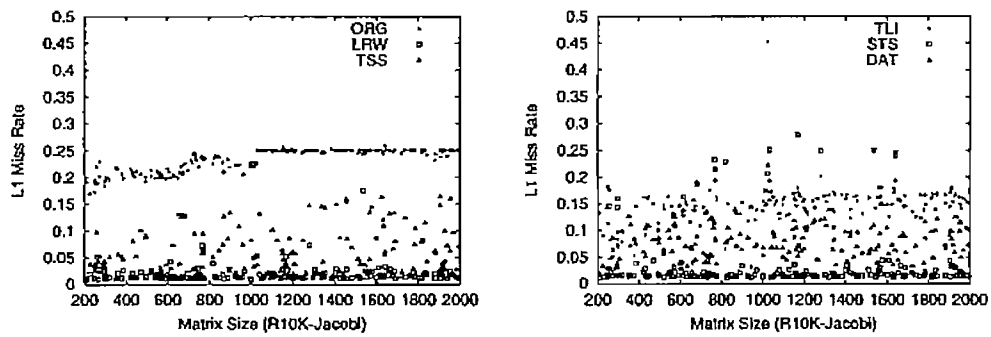



Figure 18: L1 cache miss rate of Jacobi for various schemes on the R10K

Table 3: Speedup by STS and average cache miss rates for different schemes for SOR

| Ultra II | ORG | LRW | TSS | TLI | STS | DAT |
|---|---|---|---|---|---|---|
| Average Speedup by STS | 1.10 | 1.06 | 1.34 | 1.03 | 1.00 | 1.10 |
| L1 Miss Rate | 0.14 | 0.02 | 0.07 | 0.03 | 0.02 | 0.06 |
| L2 Miss Rate | 0.066 | 0.006 | 0.009 | 0.005 | 0.006 | 0.008 |
| R10K | ORG | LRW | TSS | TLI | STS | DAT |
| Average Speedup by STS | 1.26 | 0.99 | 1.06 | 0.98 | 1.00 | 0.97 |
| L1 Miss Rate | 0.113 | 0.006 | 0.024 | 0.012 | 0.008 | 0.031 |
| L2 Miss Rate | 0.116 | 0.057 | 0.030 | 0.031 | 0.085 | 0.007 |

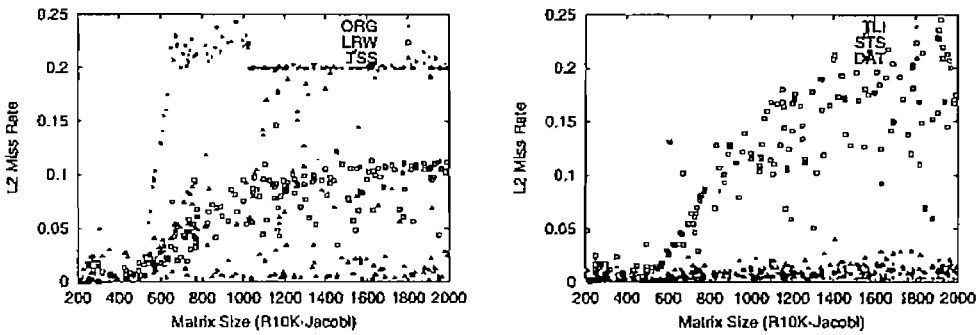



Figure 19: L2 cache miss rate of Jacobi for various schemes on the R10K

The Jacobi Kernel

We fix $ITMAX$ to 500 and randomly choose 200 array sizes ranging from 200 to 2000. The skewing factors are $S_1 = S_2 = 1$. We have $n_1 = n_3 = 17$, $n_4 = 28$ and $n_5 = 10$ on the R10K and $n_1 = n_3 = 28$, $n_4 = 24$, $n_5 = 3$ on the Ultra II. Table 4 shows the average speedup by STS, average L1 and L2 cache miss rates for Jacobi on both the Ultra II and the R10K. Specifically, Figures 14 and 17 show the execution time of Jacobi for various schemes on the Ultra II and on the R10K respectively. Figures 15 and 16 show the L1 cache and L2 cache miss rates respectively on the Ultra II. Figures 18 and 19 show the L1 cache and L2 cache miss rates respectively on the R10K.

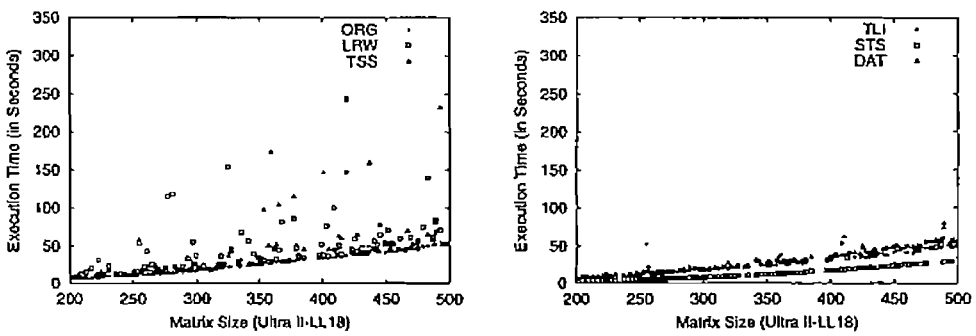



Figure 20: Execution time of LL18 for various schemes on the Ultra II

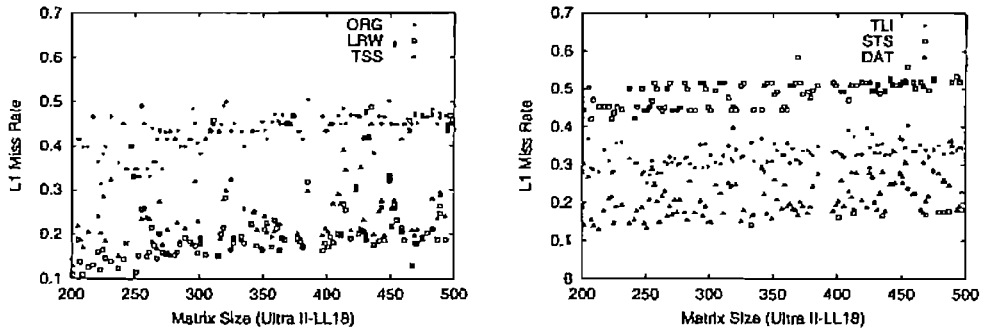



Figure 21: L1 cache miss rate of LL18 for various schemes on the Ultra II

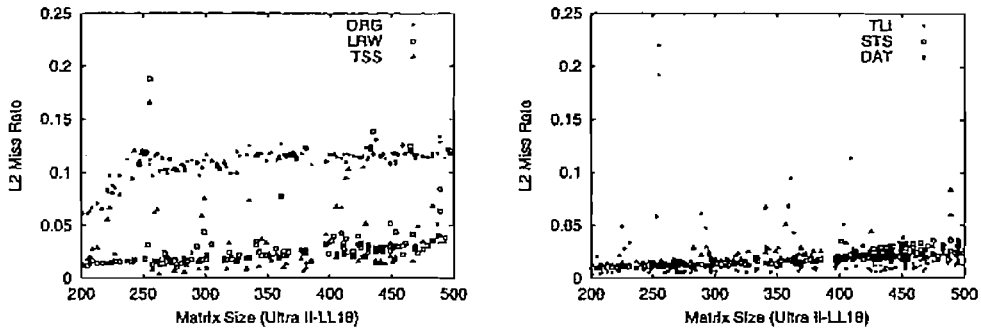



Figure 22: L2 cache miss rate of LL18 for various schemes on the Ultra II

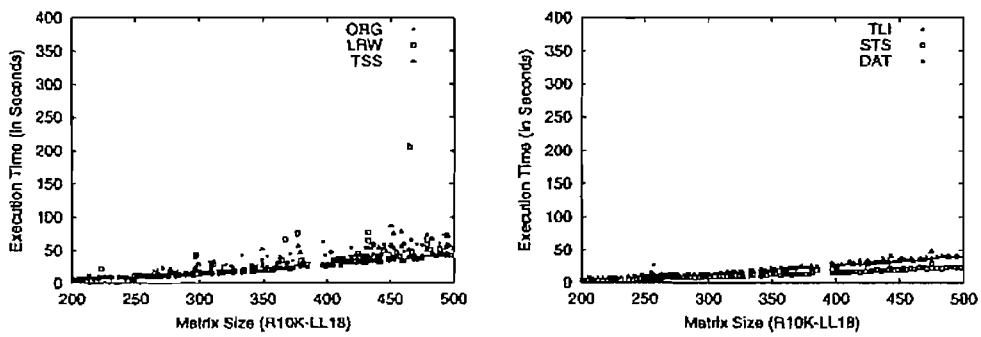



Figure 23: Execution time of LL18 for various schemes on the R10K

Table 4: Speedup by STS and average cache miss rates for different schemes for Jacobi

| Ultra II | ORG | LRW | TSS | TLI | STS | DAT |
|---|---|---|---|---|---|---|
| Speedup by STS | 5.40 | 1.39 | 2.17 | 1.28 | 1.00 | 1.10 |
| L1 Miss Rate | 0.60 | 0.12 | 0.24 | 0.24 | 0.06 | 0.19 |
| L2 Miss Rate | 0.15 | 0.02 | 0.02 | 0.01 | 0.02 | 0.01 |
| R10K | ORG | LRW | TSS | TLI | STS | DAT |
| Speedup by STS | 5.46 | 0.98 | 1.21 | 1.15 | 1.00 | 0.97 |
| L1 Miss Rate | 0.234 | 0.022 | 0.062 | 0.144 | 0.038 | 0.082 |
| L2 Miss Rate | 0.169 | 0.066 | 0.043 | 0.006 | 0.104 | 0.010 |

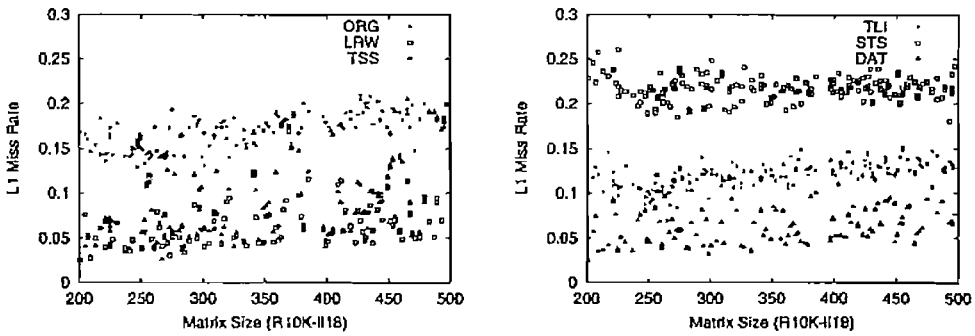



Figure 24: L1 cache miss rate of LL18 for various schemes on the R10K

the LL18 Kernel

LL18 has 9 arrays, and the tiled version has 11 arrays after duplicating $ZR$ and $ZZ$. Due to the relatively large number of arrays, the array sizes we used in SOR will produce extremely small tile sizes for all the tile-size selection schemes. Therefore, we reduce the array sizes and randomly choose 200 array sizes ranging from 200 to 500. We fix $ITMAX$ to 300. The skewing factors are $S_1 = S_2 = 2$. We have $n_1 = n_3 = 75$, $n_4 = 100$ and $n_5 = 35$ on the R10K and $n_1 = n_3 = 87$, $n_4 = 14$, $n_5 = 8$ on the Ultra II. Table 5 shows the average speedup by STS, average L1 and L2 cache miss rates for LL18 on both the Ultra II and the R10K. Specifically, Figures 20 and 23 show the execution time of LL18 for various schemes on the Ultra II and on the R10K respectively. Figures 21 and 22 show the L1 cache and L2 cache miss rates respectively on the Ultra II. Figures 24 and 25 show the L1 cache and L2 cache miss rates respectively on the R10K. Out of 200 cases,

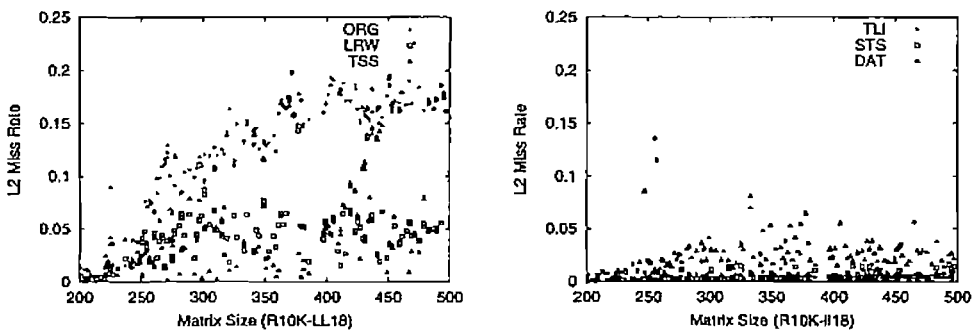



Figure 25: L2 cache miss rate of LL18 for various schemes on the R10K

Table 5: Speedup by STS and average cache miss rates for different schemes for LL18

| Ultra II | ORG | LRW | TSS | TLI | STS | DAT |
|---|---|---|---|---|---|---|
| Speedup by STS | 1.89 | 2.92 | 2.54 | 1.96 | 1.00 | 2.11 |
| L1 Miss Rate | 0.435 | 0.217 | 0.284 | 0.326 | 0.469 | 0.208 |
| L2 Miss Rate | 0.112 | 0.037 | 0.056 | 0.019 | 0.018 | 0.021 |
| R10K | ORG | LRW | TSS | TLI | STS | DAT |
| Speedup by STS | 1.72 | 1.98 | 1.98 | 1.62 | 1.00 | 1.69 |
| L1 Miss Rate | 0.173 | 0.072 | 0.096 | 0.122 | 0.217 | 0.066 |
| L2 Miss Rate | 0.128 | 0.049 | 0.075 | 0.010 | 0.005 | 0.026 |

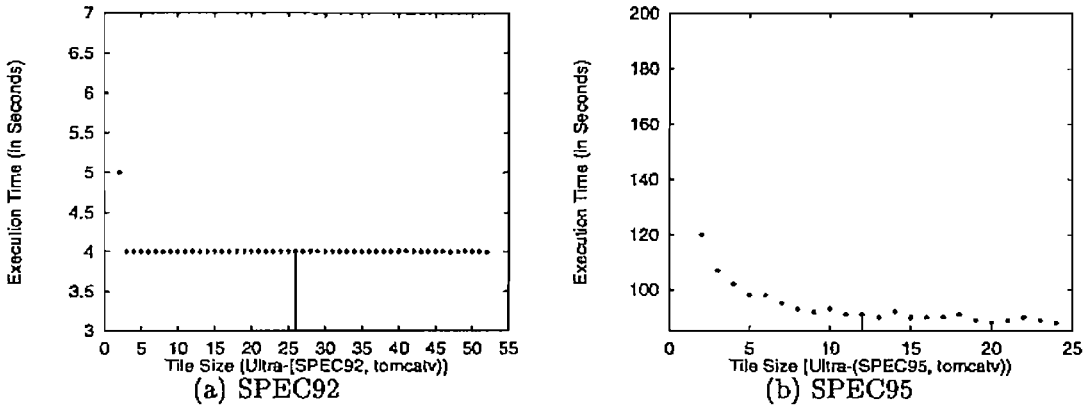



Figure 26: Performance of tomcatv with different tile sizes on the Ultra II

STS chooses 1-D tiling on 186 cases on the Ultra II and on all 200 cases on the R10K. All the other tiling schemes either choose 2-D tiling or no tiling if they fail to generate the legal tile sizes. Figures 20 and 23 indirectly show that STS can make correct selection between 1-D tiling and 2-D tiling.

`tomcatv`

`tomcatv` can only be tiled with one dimension [18], hence only STS can be applied for tile-size selection. We use two different reference inputs from SPEC92 and SPEC95 respectively. To verify whether STS produces nearly the best results, we run through a range of tile sizes, from 2 to twice of the size selected by STS, for each version of `tomcatv`. Figures 26(a) and (b) show the results on the Ultra II, where the vertical bar indicates the tile size selected by the STS. The original programs from SPEC92 and SPEC95 run 5 and 174 seconds respectively on the Ultra II, and 4.0 and 115.0 seconds respectively on the R10K. Figures 28(a) and (b) show the results on the R10K. STS chooses the near optimal tile sizes for both versions of the codes on both machines. To examine how padding will affect the STS, we also run both versions of `tomcatv` on both machines without padding applied. Figures 27(a) and (b) show the results on the Ultra II, and Figures 29(a) and (b) show the results on the R10K. Except few cases, padded version runs significantly faster than unpadded version, which demonstrates the effectiveness of padding for STS.

`swim`

Similar to `tomcatv`, `swim` is tiled only with one dimension. We use three different reference inputs from SPEC92, SPEC95 and SPEC2000 respectively. Similar to `tomcatv`, we choose the tile sizes

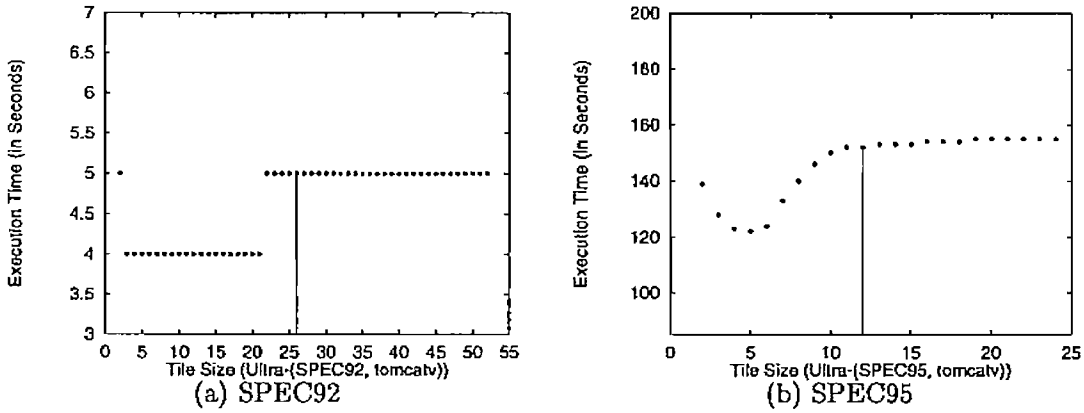



Figure 27: Performance of tomcatv with different tile sizes and without padding on the Ultra II

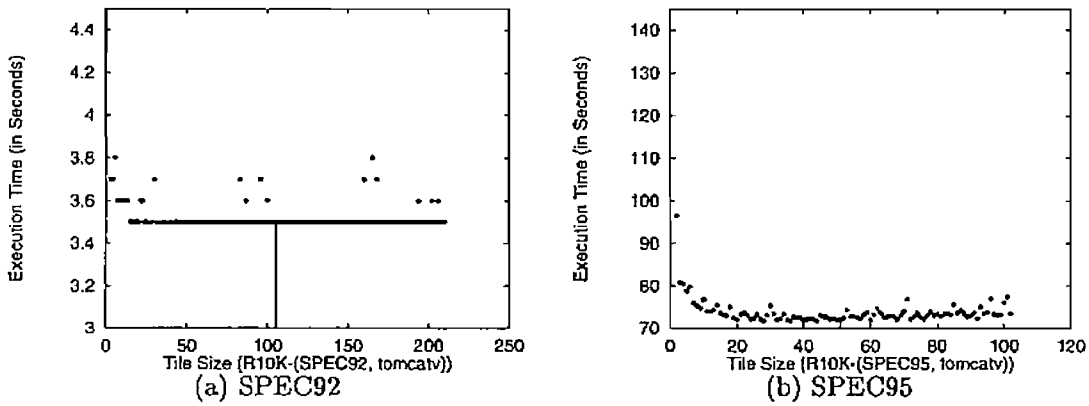



Figure 28: Performance of tomcatv with different tile sizes on the R10K

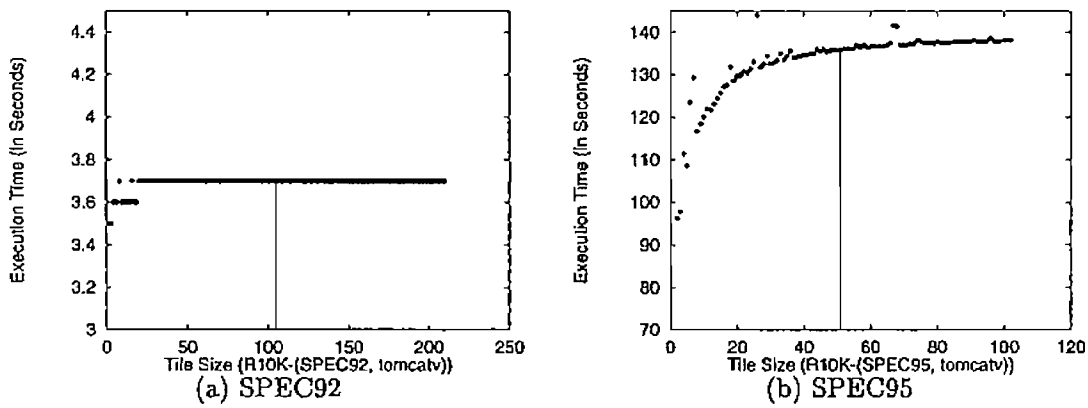
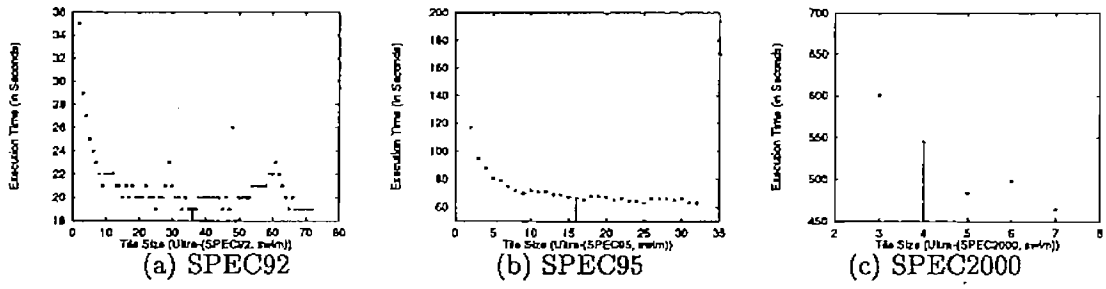
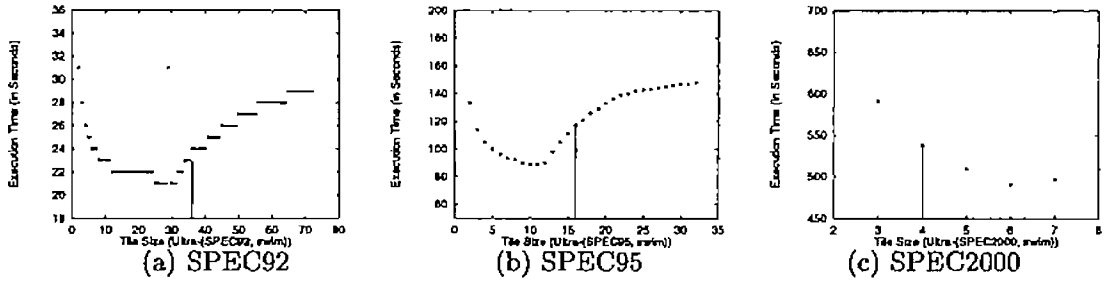



Figure 29: Performance of tomcatv with different tile sizes and without padding on the R10K



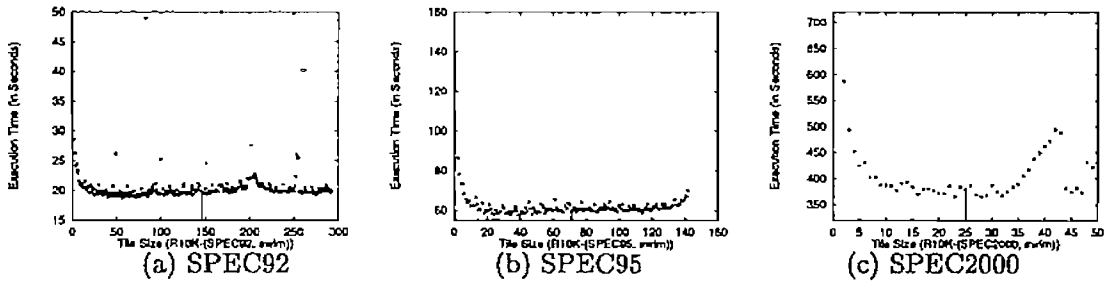

(a) SPEC92 (b) SPEC95 (c) SPEC2000

Figure 30: Performance of swim with different tile sizes on the Ultra II



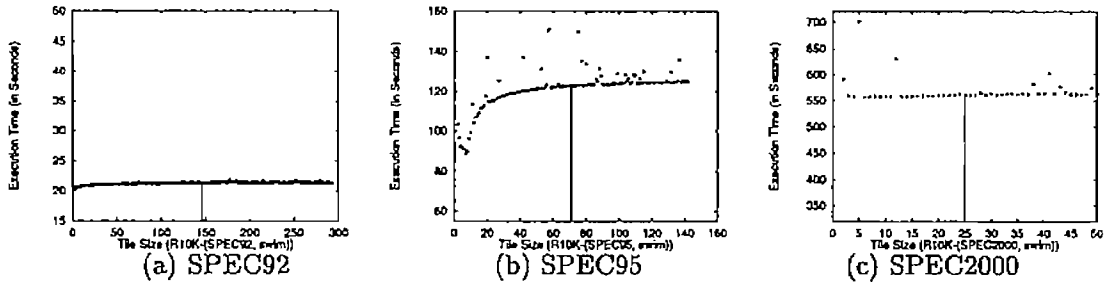

(a) SPEC92 (b) SPEC95 (c) SPEC2000

Figure 31: Performance of swim with different tile sizes and without padding on the Ultra II




(a) SPEC92 (b) SPEC95 (c) SPEC2000

Figure 32: Performance of swim with different tile sizes on the R10K




(a) SPEC92 (b) SPEC95 (c) SPEC2000

Figure 33: Performance of swim with different tile sizes and without padding on the R10K

Table 6: Summary of speedup of STS over other schemes

| | ORG | LRW | TSS | TLI | DAT |
|---|---|---|---|---|---|
| Ultra II | 2.24 | 1.63 | 1.95 | 1.37 | 1.37 |
| R10K | 2.28 | 1.24 | 1.36 | 1.22 | 1.17 |
| Both | 2.26 | 1.42 | 1.63 | 1.29 | 1.27 |

from 2 to twice of the size selected by STS for each version of swim. The original programs from SPEC92, SPEC95 and SPEC2000 run 36, 157 and 930 seconds respectively on the Ultra II, and 21.2, 91.9 and 619.5 seconds respectively on the R10K. Figures 30(a), (b) and (c) show the results on the Ultra II, and Figures 32(a), (b) and (c) show the results on the R10K. STS chooses the near optimal tile sizes for all versions of the codes on both machines. Figures 31(a), (b) and (c) show the results on the Ultra II for unpadded versions of swim, and Figures 33(a), (b) and (c) show the results on the R10K. Similar to tomcatv, padded version runs faster than unpadded version in most cases for SPEC92 and SPEC95. Note that on the Ultra II, the TLB size is smaller than the L2 cache size, hence STS will result in an underutilization of L2 cache. For SPEC2000, however, such an underutilization seems a negative effect on performance.

### 6.1 Discussion

In summary, Table 6 shows the speedup by STS over all the other schemes for all 600 cases for SOR, Jacobi and LL18, where "Both" stands for both the Ultra II and the R10K.

One interesting point is related with LRW. Considering the combination of each benchmark (SOR, Jacobi and LL18) and each machine (Ultra II and R10K), LRW produces equal or smaller average L1 cache misses in 5 out of 6 combinations compared with STS. However, this does not translate into large performance saving. (The worst average speed ratio of STS over LRW is 0.98.) We found that in general LRW produces smaller tile sizes than STS, which potentially introduces more loop overhead. For LL18, LRW has greater average L2 cache miss rates than STS since STS exploits locality for L2 cache in most of cases due to large number of arrays.

## 7 Conclusion

In this paper, we present a memory cost model to predict the cache misses after skewed tiling. Further, we model the execution cost by considering both the cache misses and the loop overhead, based on which we make a decision between tiling one loop level vs. two loop levels. We present Algorithm STS, which selects the tile size such that the capacity misses and self-interference misses within a tile traversal are eliminated. STS uses inter-array padding to eliminate cross-interference misses. We also compare STS with four previous algorithms, TLI, TSS, LRW and DAT. Experiments show that STS achieves an average speedup of 1.27 to 1.63 over all the other four algorithms. We have previously implemented a cost model along with a number of tiling algorithms [18]. However, we are yet to implement the cost model presented in this paper. Ideally, our cost model should be incorporated in a backend compiler, which will be our future work.

## References

[1] Vicki Allan, Reese Jones, Randall Lee, and Stephen Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, September 1993.

[2] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. In *Proceedings of ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 53–65, White Plains, New York, June 1990.

[3] Jacqueline Chame and Sungdo Moon. A tile selection algorithm for data locality and cache interference. In *Proceedings of the thirteenth ACM International Conference on Supercomputing*, pages 492–499, Rhodes, Greece, June 1999.

[4] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 279–290, La Jolla, CA, June 1995.

[5] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In *Proceedings of 4th International Workshop on Languages and Compilers for Parallel Computing*, August 1991. Also in *Lecture Notes in Computer Science*, pp. 328-341, Springer-Verlag, Aug. 1991.

[6] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the 8th ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 228–239, San Jose, California, October 1998.

[7] John Hennessy and David Patterson. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann Publishers, 1996.

[8] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. Data-centric multi-level blocking. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 346–357, Las Vegas, NV, June 1997.

[9] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Santa Clara, CA, April 1991.

[10] Naraig Manjikian and Tarek Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Transactions on Parallel and Distributed Systems*, 8(2):193–209, February 1997.

[11] Karhryn McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[12] Nicholas Mitchell, Karin Högstedt, Larry Carter, and Jeanne Ferrante. Quantifying the multi-level nature of tiling interactions. *International Journal of Parallel Programming*, 26(6):641–670, December 1998.

[13] Preeti Panda, Hiroshi Nakamura, Nikil Dutt, and Alexandru Nicolau. Augmenting loop tiling with data alignment for improved cache performance. *IEEE Transactions on Computers*, 48(2):142–149, February 1999.

[14] Stephen Park and Keith Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, October 1988.

[15] Gabriel Rivera and Chau-Wen Tseng. Eliminating conflict misses for high performance architectures. In *Proceedings of the 1998 ACM International Conference on Supercomputing*, pages 353–360, Melbourne, Australia, July 1998.

[16] Gabriel Rivera and Chau-Wen Tseng. A comparison of compiler tiling algorithms. In *Proceedings of the 8th International Conference on Compiler Construction*, Amsterdam, The Netherlands, March 1999.

[17] Yonghong Song and Zhiyuan Li. A compiler framework for tiling imperfectly-nested loops. In *the 12th International Workshop on Languages and Compilers for Parallel Computing*, San Diego, CA, August 1999.

[18] Yonghong Song and Zhiyuan Li. New tiling techniques to improve cache temporal locality. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 215–228, Atlanta, GA, May 1999.

[19] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proceedings of SIGMETRICS'94*, pages 261–271, Santa Clara, CA, 1994.

[20] Michael Wolf. *Improving Locality and Parallelism in Nested Loops.* PhD thesis, Department of Computer Science, Stanford University, August 1992.

[21] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 30–44, Toronto, Ontario, Canada, June 1991.

[22] Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 274–286, Paris, France, December 1996.

[23] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, 1995.